27장 배열

27.1 배열이란?

여러 개의 값을 순차적으로 나열한 자료구조.

사용 빈도가 매우 높은 가장 기본적인 자료구조.

```
const arr = ['apple', 'banana', 'orange'];
```

요소 (Element)

배열이 가지고 있는 값.

인덱스 (Index)

배열에서 배열의 요소의 위치 (0 이상의 정수)

대괄호 표기법을 통해 요소에 접근.

```
arr[0] // -> 'apple'
arr[1] // -> 'banana'
arr[2] // -> 'orange'
```

length 프로퍼티

배열의 길이를 나타내는 프로퍼티.

```
arr.length // -> 3
```

```
// 배열의 순회
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]); // 'apple' 'banana' 'orange'
}
```

배열 리터럴, Array 생성자 함수, Array.of, Array.from 메서드로 생성 가능.

```
const arr = [1, 2, 3];
```

```
arr.constructor === Array // -> true
Object.getPrototypeOf(arr) === Array.prototype // -> true
```

자바스크립트에서 배열은 객체 타입.

```
typeof arr // -> object
```

하지만 "값의 순서"와 "length 프로퍼티"에서 둘은 명확한 차이가 있다.

```
const arr = [1, 2, 3];

// 반복문으로 자료 구조를 순서대로 순회하기 위해서는 자료 구조의 요소에 순서대로
// 접근할 수 있어야 하며 자료 구조의 길이를 알 수 있어야 한다.
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]); // 1 2 3
}
```

27.2 자바스크립트 배열은 배열이 아니다

- 밀집 배열 (dense array)

  배열의 요소가 하나의 데이터 타입으로 통일되어 있고 연속적으로 인접.

인덱스를 통해 단 한번의 연산으로 특정 요소에 접근 가능. (`O(1)`)

정렬되지 않은 배열의 경우 처음 위치부터 선형적으로 검색해야 한다. (`O(n)`)

```
// 선형 검색을 통해 배열(array)에 특정 요소(target)가 존재하는지 확인한다.
// 배열에 특정 요소가 존재하면 특정 요소의 인덱스를 반환하고, 존재하지 않으면 -1을 반환한다.
function linearSearch(array, target) {
  const length = array.length;

  for (let i = 0; i < length; i++) {
    if (array[i] === target) return i;
  }
```

```
  return -1;
}

console.log(linearSearch([1, 2, 3, 4, 5, 6], 3)); // 2
console.log(linearSearch([1, 2, 3, 4, 5, 6], 0)); // -1
```

- 희소 배열 (sparse array)

  배열 요소 각각의 메모리 공간은 서로 다를 수 있고 연속적이지 않을 수 있다.

자바스크립트의 배열은 일반적인 배열을 흉내낸 특수한 객체.

```
// "16.2. 프로퍼티 어트리뷰트와 프로퍼티 디스크립터 객체" 참고
console.log(Object.getOwnPropertyDescriptors([1, 2, 3]));
/*
{
  '0': {value: 1, writable: true, enumerable: true, configu
rable: true}
  '1': {value: 2, writable: true, enumerable: true, configu
rable: true}
  '2': {value: 3, writable: true, enumerable: true, configu
rable: true}
  length: {value: 3, writable: true, enumerable: false, con
figurable: false}
}
*/
```

```
const arr = [
  'string',
  10,
  true,
  null,
  undefined,
  NaN,
  Infinity,
  [ ],
  { },
```

```
  function () {}
];
```

27.3 length 프로퍼티와 희소 배열

배열의 길이를 나타내는 0 이상의 정수.

```
[].length        // -> 0
[1, 2, 3].length // -> 3
```

현재 길이보다 더 작은 값을 length 프로퍼티에 할당하면 길이가 줄어든다.

```
const arr = [1, 2, 3, 4, 5];

// 현재 length 프로퍼티 값인 5보다 작은 숫자 값 3을 length 프로퍼티에 할당
arr.length = 3;

// 배열의 길이가 5에서 3으로 줄어든다.
console.log(arr); // [1, 2, 3]
```

```
const arr = [1, 2, 3];
console.log(arr.length); // 3

// 요소 추가
arr.push(4);
// 요소를 추가하면 length 프로퍼티의 값이 자동 갱신된다.
console.log(arr.length); // 4

// 요소 삭제
arr.pop();
// 요소를 삭제하면 length 프로퍼티의 값이 자동 갱신된다.
console.log(arr.length); // 3
```

```
// 희소 배열
const sparse = [, 2, , 4];
```

```
// 희소 배열의 length 프로퍼티 값은 요소의 개수와 일치하지 않는다.
console.log(sparse.length); // 4
console.log(sparse); // [empty, 2, empty, 4]

// 배열 sparse에는 인덱스가 0, 2인 요소가 존재하지 않는다.
console.log(Object.getOwnPropertyDescriptors(sparse));
/*
{
  '1': { value: 2, writable: true, enumerable: true, config
urable: true },
  '3': { value: 4, writable: true, enumerable: true, config
urable: true },
  length: { value: 4, writable: true, enumerable: false, co
nfigurable: false }
}
*/
```

희소 배열은 length와 배열 요소 개수가 일치하지 않으며, length가 항상 더 크다.

27.4 배열 생성

27.4.1 배열 리터럴

가장 일반적이고 간편한 배열 생성 방식.

```
const arr = [1, 2, 3];
console.log(arr.length); // 3
```

```
const arr = [];
console.log(arr.length); // 0
```

```
const arr = [1, , 3]; // 희소 배열

// 희소 배열의 length는 배열의 실제 요소 개수보다 언제나 크다.
console.log(arr.length); // 3
```

```
console.log(arr);        // [1, empty, 3]
console.log(arr[1]);     // undefined
```

27.4.2 Array 생성자 함수

```
const arr = new Array(10);

console.log(arr); // [empty × 10]
console.log(arr.length); // 10
```

```
console.log(Object.getOwnPropertyDescriptors(arr));
/*
{
  length: {value: 10, writable: true, enumerable: false, co
nfigurable: false}
}
*/
```

```
// 배열은 요소를 최대 4,294,967,295개 가질 수 있다.
new Array(4294967295);

// 전달된 인수가 0 ~ 4,294,967,295를 벗어나면 RangeError가 발생한
다.
new Array(4294967296); // RangeError: Invalid array length

// 전달된 인수가 음수이면 에러가 발생한다.
new Array(-1); // RangeError: Invalid array length
```

```
new Array(); // -> []
```

```
// 전달된 인수가 2개 이상이면 인수를 요소로 갖는 배열을 생성한다.
new Array(1, 2, 3); // -> [1, 2, 3]

// 전달된 인수가 1개지만 숫자가 아니면 인수를 요소로 갖는 배열을 생성한
```

```
다.
new Array({}); // -> [{}]
```

```
Array(1, 2, 3); // -> [1, 2, 3]
```

27.4.3 Array.of

전달된 인수를 요소로 갖는 배열을 생성한다.

```
// 전달된 인수가 1개이고 숫자이더라도 인수를 요소로 갖는 배열을 생성한
다.
Array.of(1); // -> [1]

Array.of(1, 2, 3); // -> [1, 2, 3]

Array.of('string'); // -> ['string']
```

27.4.4 Array.from

유사 배열 객체 또는 이터러블 객체를 인수로 전달받아 배열 생성.

```
// 유사 배열 객체를 변환하여 배열을 생성한다.
Array.from({ length: 2, 0: 'a', 1: 'b' }); // -> ['a', 'b']

// 이터러블을 변환하여 배열을 생성한다. 문자열은 이터러블이다.
Array.from('Hello'); // -> ['H', 'e', 'l', 'l', 'o']
```

```
// Array.from에 length만 존재하는 유사 배열 객체를 전달하면 undefi
ned를 요소로 채운다.
Array.from({ length: 3 }); // -> [undefined, undefined, und
efined]

// Array.from은 두 번째 인수로 전달한 콜백 함수의 반환값으로 구성된
배열을 반환한다.
Array.from({ length: 3 }, (_, i) => i); // -> [0, 1, 2]
```

27.5 배열 요소의 참조

대괄호에 인덱스를 넣어 배열의 요소를 참조한다.

```
const arr = [1, 2];

// 인덱스가 0인 요소를 참조
console.log(arr[0]); // 1
// 인덱스가 1인 요소를 참조
console.log(arr[1]); // 2
```

```
const arr = [1, 2];

// 인덱스가 2인 요소를 참조. 배열 arr에는 인덱스가 2인 요소가 존재하지 않는다.
console.log(arr[2]); // undefined
```

```
// 희소 배열
const arr = [1, , 3];

// 배열 arr에는 인덱스가 1인 요소가 존재하지 않는다.
console.log(Object.getOwnPropertyDescriptors(arr));
/*
{
  '0': {value: 1, writable: true, enumerable: true, configu
rable: true},
  '2': {value: 3, writable: true, enumerable: true, configu
rable: true},
  length: {value: 3, writable: true, enumerable: false, con
figurable: false}
*/

// 존재하지 않는 요소를 참조하면 undefined가 반환된다.
console.log(arr[1]); // undefined
console.log(arr[3]); // undefined
```

27.6 배열 요소의 추가와 갱신

존재하지 않는 인덱스를 통해 값을 할당하면 새로운 요소가 추가된다.

동시에 length 프로퍼티의 값이 자동 갱신된다.

```
const arr = [0];

// 배열 요소의 추가
arr[1] = 1;

console.log(arr); // [0, 1]
console.log(arr.length); // 2
```

```
arr[100] = 100;

console.log(arr); // [0, 1, empty × 98, 100]
console.log(arr.length); // 101
```

```
// 명시적으로 값을 할당하지 않은 요소는 생성되지 않는다.
console.log(Object.getOwnPropertyDescriptors(arr));
/*
{
  '0': {value: 0, writable: true, enumerable: true, configu
rable: true},
  '1': {value: 1, writable: true, enumerable: true, configu
rable: true},
  '100': {value: 100, writable: true, enumerable: true, con
figurable: true},
  length: {value: 101, writable: true, enumerable: false, c
onfigurable: false}
*/
```

```
// 요소값의 갱신
arr[1] = 10;

console.log(arr); // [0, 10, empty × 98, 100]
```

27.7 배열 요소의 삭제

배열은 객체이기 때문에 delete 연산자를 사용할 수도 있다.

```
const arr = [1, 2, 3];

// 배열 요소의 삭제
delete arr[1];
console.log(arr); // [1, empty, 3]

// length 프로퍼티에 영향을 주지 않는다. 즉, 희소 배열이 된다.
console.log(arr.length); // 3
```

length 프로퍼티에 영향이 없고 희소 배열이 되기 때문에 사용하지 않는 것이 좋다.

Array.prototype.splice 메서드를 사용하는 것이 좋다.'

```
const arr = [1, 2, 3];

// Array.prototype.splice(삭제를 시작할 인덱스, 삭제할 요소 수)
// arr[1]부터 1개의 요소를 제거
arr.splice(1, 1);
console.log(arr); // [1, 3]

// length 프로퍼티가 자동 갱신된다.
console.log(arr.length); // 2
```

27.8 배열 메서드

원본 배열을 직접 변경하는 메서드 (mutator method)

원본 배열을 직접 변경하지 않고 새로운 배열을 생성하여 반환하는 메서드 (accessor method)

```
const arr = [1];

// push 메서드는 원본 배열(arr)을 직접 변경한다.
arr.push(2);
console.log(arr); // [1, 2]
```

```
// concat 메서드는 원본 배열(arr)을 직접 변경하지 않고 새로운 배열을
생성하여 반환한다.
const result = arr.concat(3);
console.log(arr);    // [1, 2]
console.log(result); // [1, 2, 3]
```

후자를 사용하는 것이 좋다.

27.8.1 Array.isArray

전달된 인수가 배열인지 여부를 boolean으로 반환.

```
// true
Array.isArray([]);
Array.isArray([1, 2]);
Array.isArray(new Array());

// false
Array.isArray();
Array.isArray({});
Array.isArray(null);
Array.isArray(undefined);
Array.isArray(1);
Array.isArray('Array');
Array.isArray(true);
Array.isArray(false);
Array.isArray({ 0: 1, length: 1 })
```

27.8.2 Array.prototype.indexOf

원본 배열에서 인수로 전달된 요소의 인덱스를 반환.

```
const arr = [1, 2, 2, 3];

// 배열 arr에서 요소 2를 검색하여 첫 번째로 검색된 요소의 인덱스를 반환
한다.
arr.indexOf(2);    // -> 1
// 배열 arr에 요소 4가 없으므로 -1을 반환한다.
```

```
arr.indexOf(4);    // -> -1
// 두 번째 인수는 검색을 시작할 인덱스다. 두 번째 인수를 생략하면 처음부터 검색한다.
arr.indexOf(2, 2); // -> 2
```

```
const foods = ['apple', 'banana', 'orange'];

// foods 배열에 'orange' 요소가 존재하는지 확인한다.
if (foods.indexOf('orange') === -1) {
  // foods 배열에 'orange' 요소가 존재하지 않으면 'orange' 요소를 추가한다.
  foods.push('orange');
}

console.log(foods); // ["apple", "banana", "orange"]
```

```
const foods = ['apple', 'banana'];

// foods 배열에 'orange' 요소가 존재하는지 확인한다.
if (!foods.includes('orange')) {
  // foods 배열에 'orange' 요소가 존재하지 않으면 'orange' 요소를 추가한다.
  foods.push('orange');
}

console.log(foods); // ["apple", "banana", "orange"]
```

27.8.3 Array.prototype.push

인수를 배열의 마지막 요소로 추가한다.

```
const arr = [1, 2];

// 인수로 전달받은 모든 값을 원본 배열 arr의 마지막 요소로 추가하고 변경된 length 값을 반환한다.
let result = arr.push(3, 4);
console.log(result); // 4
```

```
// push 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 2, 3, 4]
```

push 메서드 보다는 length 프로퍼티로 마지막에 직접 추가하는 것이 더 성능적으로 좋다.

```
const arr = [1, 2];

// arr.push(3)과 동일한 처리를 한다. 이 방법이 push 메서드보다 빠르
다.
arr[arr.length] = 3;
console.log(arr); // [1, 2, 3]
```

push 메서드는 원본 배열을 직접 변경하므로 ES6 스프레드 문법을 사용하는 것이 좋다.

```
const arr = [1, 2];

// ES6 스프레드 문법
const newArr = [...arr, 3];
console.log(newArr); // [1, 2, 3]
```

27.8.4 Array.prototype.pop

원본 배열에서 마지막 요소를 제거하고 제거한 요소를 반환.

빈 배열이면 undefined를 반환하며, 원본 배열을 직접 변경한다.

```
const arr = [1, 2];

// 원본 배열에서 마지막 요소를 제거하고 제거한 요소를 반환한다.
let result = arr.pop();
console.log(result); // 2

// pop 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1]
```

push와 pop을 통해 스택을 구현할 수 있다.

```
const Stack = (function () {
  function Stack(array = []) {
    if (!Array.isArray(array)) {
      // "47. 에러 처리" 참고
      throw new TypeError(`${array} is not an array.`);
    }
    this.array = array;
  }

  Stack.prototype = {
    // "19.10.1. 생성자 함수에 의한 프로토타입의 교체" 참고
    constructor: Stack,
    // 스택의 가장 마지막에 데이터를 밀어 넣는다.
    push(value) {
      return this.array.push(value);
    },
    // 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신 데이터
를 꺼낸다.
    pop() {
      return this.array.pop();
    },
    // 스택의 복사본 배열을 반환한다.
    entries() {
      return [...this.array];
    }
  };

  return Stack;
}());

const stack = new Stack([1, 2]);
console.log(stack.entries()); // [1, 2]

stack.push(3);
console.log(stack.entries()); // [1, 2, 3]
```

```
stack.pop();
console.log(stack.entries()); // [1, 2]
```

```
class Stack {
  #array; // private class member

  constructor(array = []) {
    if (!Array.isArray(array)) {
      throw new TypeError(`${array} is not an array.`);
    }
    this.#array = array;
  }

  // 스택의 가장 마지막에 데이터를 밀어 넣는다.
  push(value) {
    return this.#array.push(value);
  }

  // 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신 데이터를
꺼낸다.
  pop() {
    return this.#array.pop();
  }

  // 스택의 복사본 배열을 반환한다.
  entries() {
    return [...this.#array];
  }
}

const stack = new Stack([1, 2]);
console.log(stack.entries()); // [1, 2]

stack.push(3);
console.log(stack.entries()); // [1, 2, 3]
```

```
stack.pop();
console.log(stack.entries()); // [1, 2]
```

27.8.5 Array.prototype.unshift

인수를 원본 배열의 선두에 추가.

원본 배열을 직접 변경.

```
const arr = [1, 2];

// 인수로 전달받은 모든 값을 원본 배열의 선두에 요소로 추가하고 변경된 l
ength 값을 반환한다.
let result = arr.unshift(3, 4);
console.log(result); // 4

// unshift 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [3, 4, 1, 2]
```

역시 ES6의 스프레드 문법을 사용하는 것이 좋다.

```
const arr = [1, 2];

// ES6 스프레드 문법
const newArr = [3, ...arr];
console.log(newArr); // [3, 1, 2]
```

27.8.6 Array.prototype.shift

원본 배열에서 첫 번째 요소를 제거하고 제거된 요소를 반환.

원본 배열을 직접 변경.

```
const arr = [1, 2];

// 원본 배열에서 첫 번째 요소를 제거하고 제거한 요소를 반환한다.
let result = arr.shift();
console.log(result); // 1
```

```
// shift 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [2]
```

shift와 push를 통해 큐를 규현할 수 있다.

```
const Queue = (function () {
  function Queue(array = []) {
    if (!Array.isArray(array)) {
      // "47. 에러 처리" 참고
      throw new TypeError(`${array} is not an array.`);
    }
    this.array = array;
  }

  Queue.prototype = {
    // "19.10.1. 생성자 함수에 의한 프로토타입의 교체" 참고
    constructor: Queue,
    // 큐의 가장 마지막에 데이터를 밀어 넣는다.
    enqueue(value) {
      return this.array.push(value);
    },
    // 큐의 가장 처음 데이터, 즉 가장 먼저 밀어 넣은 데이터를 꺼낸다.
    dequeue() {
      return this.array.shift();
    },
    // 큐의 복사본 배열을 반환한다.
    entries() {
      return [...this.array];
    }
  };

  return Queue;
}());

const queue = new Queue([1, 2]);
console.log(queue.entries()); // [1, 2]

queue.enqueue(3);
```

```
console.log(queue.entries()); // [1, 2, 3]

queue.dequeue();
console.log(queue.entries()); // [2, 3]
```

```
class Queue {
  #array; // private class member

  constructor(array = []) {
    if (!Array.isArray(array)) {
      throw new TypeError(`${array} is not an array.`);
    }
    this.#array = array;
  }

  // 큐의 가장 마지막에 데이터를 밀어 넣는다.
  enqueue(value) {
    return this.#array.push(value);
  }

  // 큐의 가장 처음 데이터, 즉 가장 먼저 밀어 넣은 데이터를 꺼낸다.
  dequeue() {
    return this.#array.shift();
  }

  // 큐의 복사본 배열을 반환한다.
  entries() {
    return [...this.#array];
  }
}

const queue = new Queue([1, 2]);
console.log(queue.entries()); // [1, 2]

queue.enqueue(3);
console.log(queue.entries()); // [1, 2, 3]
```

```
queue.dequeue();
console.log(queue.entries()); // [2, 3]
```

27.8.7 Array.prototype.concat

값 또는 배열을 인수로 전달받아 원본 배열의 마지막 요소로 추가한 새로운 배열을 반환.

```
const arr1 = [1, 2];
const arr2 = [3, 4];

// 배열 arr2를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환한다.
// 인수로 전달한 값이 배열인 경우 배열을 해체하여 새로운 배열의 요소로 추가한다.
let result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4]

// 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환한다.
result = arr1.concat(3);
console.log(result); // [1, 2, 3]

// 배열 arr2와 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환한다.
result = arr1.concat(arr2, 5);
console.log(result); // [1, 2, 3, 4, 5]

// 원본 배열은 변경되지 않는다.
console.log(arr1); // [1, 2]
```

새로운 배열을 반환하며 반환값을 반드시 변수에 할당해야 한다.

```
const arr1 = [3, 4];

// unshift 메서드는 원본 배열을 직접 변경한다.
// 따라서 원본 배열을 변수에 저장해 두지 않으면 변경된 배열을 사용할 수 없다.
arr1.unshift(1, 2);
// unshift 메서드를 사용할 경우 원본 배열을 반드시 변수에 저장해 두어
```

```
야 결과를 확인할 수 있다.
console.log(arr1); // [1, 2, 3, 4]

// push 메서드는 원본 배열을 직접 변경한다.
// 따라서 원본 배열을 변수에 저장해 두지 않으면 변경된 배열을 사용할 수 없다.
arr1.push(5, 6);
// push 메서드를 사용할 경우 원본 배열을 반드시 변수에 저장해 두어야 결과를 확인할 수 있다.
console.log(arr1); // [1, 2, 3, 4, 5, 6]

// unshift와 push 메서드는 concat 메서드로 대체할 수 있다.
const arr2 = [3, 4];

// concat 메서드는 원본 배열을 변경하지 않고 새로운 배열을 반환한다.
// arr1.unshift(1, 2)를 다음과 같이 대체할 수 있다.
let result = [1, 2].concat(arr2);
console.log(result); // [1, 2, 3, 4]

// arr1.push(5, 6)를 다음과 같이 대체할 수 있다.
result = result.concat(5, 6);
console.log(result); // [1, 2, 3, 4, 5, 6]
```

```
const arr = [3, 4];

// unshift와 push 메서드는 인수로 전달받은 배열을 그대로 원본 배열의 요소로 추가한다
arr.unshift([1, 2]);
arr.push([5, 6]);
console.log(arr); // [[1, 2], 3, 4,[5, 6]]

// concat 메서드는 인수로 전달받은 배열을 해체하여 새로운 배열의 요소로 추가한다
let result = [1, 2].concat([3, 4]);
result = result.concat([5, 6]);

console.log(result); // [1, 2, 3, 4, 5, 6]
```

역시 ES6의 스프레드 문법으로 대체할 수 있다.

```
let result = [1, 2].concat([3, 4]);
console.log(result); // [1, 2, 3, 4]

// concat 메서드는 ES6의 스프레드 문법으로 대체할 수 있다.
result = [...[1, 2], ...[3, 4]];
console.log(result); // [1, 2, 3, 4]
```

push, pop, unshift 등의 메서드보다는 ES6의 스프레드 문법을 일관성있게 사용하는 것이 더 좋다.

27.8.8 Array.prototype.splice

원본 배열의 중간에 있는 요소를 추가 또는 제거할 경우 사용.

```
const arr = [1, 2, 3, 4];

// 원본 배열의 인덱스 1부터 2개의 요소를 제거하고 그 자리에 새로운 요소 20, 30을 삽입한다.
const result = arr.splice(1, 2, 20, 30);

// 제거한 요소가 배열로 반환된다.
console.log(result); // [2, 3]
// splice 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 20, 30, 4]
```

```
const arr = [1, 2, 3, 4];

// 원본 배열의 인덱스 1부터 0개의 요소를 제거하고 그 자리에 새로운 요소 100을 삽입한다.
const result = arr.splice(1, 0, 100);

// 원본 배열이 변경된다.
console.log(arr); // [1, 100, 2, 3, 4]
// 제거한 요소가 배열로 반환된다.
console.log(result); // []
```

```
const arr = [1, 2, 3, 4];

// 원본 배열의 인덱스 1부터 2개의 요소를 제거한다.
const result = arr.splice(1, 2);

// 원본 배열이 변경된다.
console.log(arr); // [1, 4]
// 제거한 요소가 배열로 반환된다.
console.log(result); // [2, 3]
```

```
const arr = [1, 2, 3, 4];

// 원본 배열의 인덱스 1부터 모든 요소를 제거한다.
const result = arr.splice(1);

// 원본 배열이 변경된다.
console.log(arr); // [1]
// 제거한 요소가 배열로 반환된다.
console.log(result); // [2, 3, 4]
```

indexOf 로 특정 요소의 인덱스를 알아내어 제거할 수 있다.

```
const arr = [1, 2, 3, 1, 2];

// 배열 array에서 item 요소를 제거한다. item 요소가 여러 개 존재하면
첫 번째 요소만 제거한다.
function remove(array, item) {
  // 제거할 item 요소의 인덱스를 취득한다.
  const index = array.indexOf(item);

  // 제거할 item 요소가 있다면 제거한다.
  if (index !== -1) array.splice(index, 1);

  return array;
}
```

```
console.log(remove(arr, 2)); // [1, 3, 1, 2]
console.log(remove(arr, 10)); // [1, 3, 1, 2]
```

중복된 특정 요소를 전부 제거하려면 filter 메서드를 사용할 수 있다.

```
const arr = [1, 2, 3, 1, 2];

// 배열 array에서 모든 item 요소를 제거한다.
function removeAll(array, item) {
  return array.filter(v => v !== item);
}

console.log(removeAll(arr, 2)); // [1, 3, 1]
```

27.8.9 Array.prototype.slice

인수로 전달된 범위의 요소들을 복사하여 배열로 반환.

복사를 시작할 인덱스 (start), 복사를 종료할 인덱스 (end)를 매개 변수로 받는다.

```
const arr = [1, 2, 3];

// arr[0]부터 arr[1] 이전(arr[1] 미포함)까지 복사하여 반환한다.
arr.slice(0, 1); // -> [1]

// arr[1]부터 arr[2] 이전(arr[2] 미포함)까지 복사하여 반환한다.
arr.slice(1, 2); // -> [2]

// 원본은 변경되지 않는다.
console.log(arr); // [1, 2, 3]
```

```
const arr = [1, 2, 3];

// arr[1]부터 이후의 모든 요소를 복사하여 반환한다.
arr.slice(1); // -> [2, 3]
```

```
const arr = [1, 2, 3];

// 배열의 끝에서부터 요소를 한 개 복사하여 반환한다.
arr.slice(-1); // -> [3]

// 배열의 끝에서부터 요소를 두 개 복사하여 반환한다.
arr.slice(-2); // -> [2, 3]
```

```
const arr = [1, 2, 3];

// 인수를 모두 생략하면 원본 배열의 복사본을 생성하여 반환한다.
const copy = arr.slice();
console.log(copy); // [1, 2, 3]
console.log(copy === arr); // false
```

```
const todos = [
  { id: 1, content: 'HTML', completed: false },
  { id: 2, content: 'CSS', completed: true },
  { id: 3, content: 'Javascript', completed: false }
];

// 얕은 복사(shallow copy)
const _todos = todos.slice();
// const _todos = [...todos];

// _todos와 todos는 참조값이 다른 별개의 객체다.
console.log(_todos === todos); // false

// 배열 요소의 참조값이 같다. 즉, 얕은 복사되었다.
console.log(_todos[0] === todos[0]); // true
```

27.8.10 Array.prototype.join

원본 배열의 요소를 문자로 변환하고 인수로 전달 받은 문자열을 구분자로 연결하여 반환.

```
const arr = [1, 2, 3, 4];

// 기본 구분자는 ','이다.
// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 기본 구분자 ','로 연결한 문자열을 반환한다.
arr.join(); // -> '1,2,3,4';

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 빈문자열로 연결한 문자열을 반환한다.
arr.join(''); // -> '1234'

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 구분자 ':'로 연결한 문자열을 반환한다.ㄴ
arr.join(':'); // -> '1:2:3:4'
```

27.8.11 Array.prototype.reverse

원본 배열의 순서를 반대로 뒤집으며 원본 배열이 변경된다.

```
const arr = [1, 2, 3];
const result = arr.reverse();

// reverse 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [3, 2, 1]
// 반환값은 변경된 배열이다.
console.log(result); // [3, 2, 1]
```

27.8.12 Array.prototype.fill

ES6에서부터 도입.

인수로 전달받은 값으로 배열의 처음부터 끝까지 변경.

```
const arr = [1, 2, 3];

// 인수로 전달 받은 값 0을 배열의 처음부터 끝까지 요소로 채운다.
arr.fill(0);
```

```
// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [0, 0, 0]
```

```
const arr = [1, 2, 3];

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 끝까지 요소로 채운다.
arr.fill(0, 1);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 0, 0]
```

```
const arr = [1, 2, 3, 4, 5];

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 3 이전(인덱스 3 미포함)
까지 요소로 채운다.
arr.fill(0, 1, 3);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 0, 0, 4, 5]
```

```
const arr = new Array(3);
console.log(arr); // [empty × 3]

// 인수로 전달받은 값 1을 배열의 처음부터 끝까지 요소로 채운다.
const result = arr.fill(1);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 1, 1]

// fill 메서드는 변경된 원본 배열을 반환한다.
console.log(result); // [1, 1, 1]
```

Array.from 메서드와 함께 사용할 경우 새로운 배열을 만들면서 요소를 채울 수 있다.

```
// 인수로 전달받은 정수만큼 요소를 생성하고 0부터 1씩 증가하면서 요소를
채운다.
```

```
const sequences = (length = 0) => Array.from({ length },
(_, i) => i);
// const sequences = (length = 0) => Array.from(new Array(l
ength), (_, i) => i);

console.log(sequences(3)); // [0, 1, 2]
```

27.8.13 Array.prototype.includes

배열 내에 특정 요소가 포함되어 있는지 확인하여 true 또는 false 반환.

```
const arr = [1, 2, 3];

// 배열에 요소 2가 포함되어 있는지 확인한다.
arr.includes(2); // -> true

// 배열에 요소 100이 포함되어 있는지 확인한다.
arr.includes(100); // -> false
```

```
const arr = [1, 2, 3];

// 배열에 요소 1이 포함되어 있는지 인덱스 1부터 확인한다.
arr.includes(1, 1); // -> false

// 배열에 요소 3이 포함되어 있는지 인덱스 2(arr.length - 1)부터 확인
한다.
arr.includes(3, -1); // -> true
```

```
[NaN].indexOf(NaN) !== -1; // -> false
[NaN].includes(NaN);       // -> true
```

27.8.14 Array.prototype.flat

인수로 전달한 깊이 만큼 재귀적으로 배열을 평탄화.

```
[1, [2, 3, 4, 5]].flat(); // -> [1, 2, 3, 4, 5]
```

```
// 중첩 배열을 평탄화하기 위한 깊이 값의 기본값은 1이다.
[1, [2, [3, [4]]]].flat();  // -> [1, 2, [3, [4]]]
[1, [2, [3, [4]]]].flat(1); // -> [1, 2, [3, [4]]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 2로 지정하여 2단계 깊이까지 평탄화한다.
[1, [2, [3, [4]]]].flat(2); // -> [1, 2, 3, [4]]
// 2번 평탄화한 것과 동일하다.
[1, [2, [3, [4]]]].flat().flat(); // -> [1, 2, 3, [4]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 Infinity로 지정하여 중첩 배열 모두를 평탄화한다.
[1, [2, [3, [4]]]].flat(Infinity); // -> [1, 2, 3, 4]
```

27.9 배열 고차 함수

> 고차 함수 (Higher-Order Function, HOF)
>
> 함수를 인수로 전달받거나 함수를 반환하는 함수.

외부 상태의 변경이나 가변 데이터를 피하고, 불변성을 지향하는 함수형 프로그래밍에 기반.

22.9.1 Array.prototype.sort

배열의 요소를 정렬.

```
const fruits = ['Banana', 'Orange', 'Apple'];

// 오름차순(ascending) 정렬
fruits.sort();

// sort 메서드는 원본 배열을 직접 변경한다.
console.log(fruits); // ['Apple', 'Banana', 'Orange']
```

```
const fruits = ['바나나', '오렌지', '사과'];
```

```
// 오름차순(ascending) 정렬
fruits.sort();

// sort 메서드는 원본 배열을 직접 변경한다.
console.log(fruits); // ['바나나', '사과', '오렌지']
```

sort와 reverse를 사용하여 내림차순 정렬 가능.

```
const fruits = ['Banana', 'Orange', 'Apple'];

// 오름차순(ascending) 정렬
fruits.sort();

// sort 메서드는 원본 배열을 직접 변경한다.
console.log(fruits); // ['Apple', 'Banana', 'Orange']

// 내림차순(descending) 정렬
fruits.reverse();

// reverse 메서드도 원본 배열을 직접 변경한다.
console.log(fruits); // ['Orange', 'Banana', 'Apple']
```

sort의 기본 정렬 순서는 유니코드 코드 포인트의 순서를 따른다.

배열의 요소를 일시적으로 문자열로 변환한 후 정렬하는데 숫자의 경우 의도치 않은 결과가 나올 수 있다.

따라서 숫자 정렬 시 정렬 순서를 정의하는 비교 함수를 인수로 전달해야 한다.

양수, 음수, 0을 반환할 수 있다.

양수 ⇒ 두 번째 인수를 우선시.

음수 ⇒ 첫 번째 인수를 우선시.

0 ⇒ 정렬하지 않는다.

```
const points = [40, 100, 1, 5, 2, 25, 10];

// 숫자 배열의 오름차순 정렬. 비교 함수의 반환값이 0보다 작으면 a를 우선하여 정렬한다.
points.sort((a, b) => a - b);
```

```
console.log(points); // [1, 2, 5, 10, 25, 40, 100]

// 숫자 배열에서 최소/최대값 취득
console.log(points[0], points[points.length]); // 1

// 숫자 배열의 내림차순 정렬. 비교 함수의 반환값이 0보다 작으면 b를 우선하여 정렬한다.
points.sort((a, b) => b - a);
console.log(points); // [100, 40, 25, 10, 5, 2, 1]

// 숫자 배열에서 최대값 취득
console.log(points[0]); // 100
```

```
const todos = [
  { id: 4, content: 'JavaScript' },
  { id: 1, content: 'HTML' },
  { id: 2, content: 'CSS' }
];

// 비교 함수. 매개변수 key는 프로퍼티 키다.
function compare(key) {
  // 프로퍼티 값이 문자열인 경우 - 산술 연산으로 비교하면 NaN이 나오므로 비교 연산을 사용한다.
  // 비교 함수는 양수/음수/0을 반환하면 되므로 - 산술 연산 대신 비교 연산을 사용할 수 있다.
  return (a, b) => (a[key] > b[key] ? 1 : (a[key] < b[key] ? -1 : 0));
}

// id를 기준으로 오름차순 정렬
todos.sort(compare('id'));
console.log(todos);
/*
[
  { id: 1, content: 'HTML' },
  { id: 2, content: 'CSS' },
  { id: 4, content: 'JavaScript' }
]
```

```
*/

// content를 기준으로 오름차순 정렬
todos.sort(compare('content'));
console.log(todos);
/*
[
  { id: 2, content: 'CSS' },
  { id: 1, content: 'HTML' },
  { id: 4, content: 'JavaScript' }
]
*/
```

27.9.2 Array.prototype.forEach

for 문을 대체할 수 잇는 고차 함수.

내부에서 반복문을 통해 자신을 호출한 배열을 순회한다.

수행해야 할 처리를 콜백 함수로 전달받아 반복적으로 호출한다.

```
const numbers = [1, 2, 3];
let pows = [];

// forEach 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함수를
반복 호출한다.
numbers.forEach(item => pows.push(item ** 2));
console.log(pows); // [1, 4, 9]
```

```
// forEach 메서드는 콜백 함수를 호출하면서 3개(요소값, 인덱스, this)
의 인수를 전달한다.
[1, 2, 3].forEach((item, index, arr) => {
  console.log(`요소값: ${item}, 인덱스: ${index}, this: ${JSO
N.stringify(arr)}`);
});
/*
요소값: 1, 인덱스: 0, this: [1,2,3]
요소값: 2, 인덱스: 1, this: [1,2,3]
```

```
요소값: 3, 인덱스: 2, this: [1,2,3]
*/
```

```
const numbers = [1, 2, 3];

// forEach 메서드는 원본 배열을 변경하지 않지만 콜백 함수를 통해 원본 배열을 변경할 수는 있다.
// 콜백 함수의 세 번째 매개변수 arr은 원본 배열 numbers를 가리킨다.
// 따라서 콜백 함수의 세 번째 매개변수 arr을 직접 변경하면 원본 배열 numbers가 변경된다.
numbers.forEach((item, index, arr) => { arr[index] = item * 2; });
console.log(numbers); // [1, 4, 9]
```

```
const result = [1, 2, 3].forEach(console.log);
console.log(result); // undefined
```

ES6의 화살표 함수와 같이 사용하면 this의 바인딩 이슈도 깔끔하게 해결하면서 사용할 수 있다.

```
class Numbers {
  numberArray = [];

  multiply(arr) {
    // 화살표 함수 내부에서 this를 참조하면 상위 스코프의 this를 그대로 참조한다.
    arr.forEach(item => this.numberArray.push(item * item));
  }
}

const numbers = new Numbers();
numbers.multiply([1, 2, 3]);
console.log(numbers.numberArray); // [1, 4, 9]
```

forEach문은 함수이기 때문에 return 으로 탈출할 수 있다.

```
[1, 2, 3].forEach(item => {
  console.log(item);
  if (item > 1) break; // SyntaxError: Illegal break statem
ent
});

[1, 2, 3].forEach(item => {
  console.log(item);
  if (item > 1) continue;
  // SyntaxError: Illegal continue statement: no surroundin
g iteration statement
});
```

27.9.3 Array.prototype.map

자신을 호출한 배열을 순회하면서 콜백 함수의 반환값들로 구성된 새로운 배열을 반환한다.

```
const numbers = [1, 4, 9];

// map 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함수를 반
복 호출한다.
// 그리고 콜백 함수의 반환값들로 구성된 새로운 배열을 반환한다.
const roots = numbers.map(item => Math.sqrt(item));

// 위 코드는 다음과 같다.
// const roots = numbers.map(Math.sqrt);

// map 메서드는 새로운 배열을 반환한다
console.log(roots);   // [ 1, 2, 3 ]
// map 메서드는 원본 배열을 변경하지 않는다
console.log(numbers); // [ 1, 4, 9 ]
```

map 메서드가 반환한 새로운 배열은 기존의 배열과 반드시 1대1 매핑된다.

```
// map 메서드는 콜백 함수를 호출하면서 3개(요소값, 인덱스, this)의 인
수를 전달한다.
[1, 2, 3].map((item, index, arr) => {
```

```
  console.log(`요소값: ${item}, 인덱스: ${index}, this: ${JSO
N.stringify(arr)}`);
  return item;
});
/*
요소값: 1, 인덱스: 0, this: [1,2,3]
요소값: 2, 인덱스: 1, this: [1,2,3]
요소값: 3, 인덱스: 2, this: [1,2,3]
*/
```

27.9.4 Array.prototype.filter

자신을 호출한 배열을 순회하면서 콜백 함수의 반환값이 true인 요소로만 구성된 새로운 배열을 반환한다.

```
const numbers = [1, 2, 3, 4, 5];

// filter 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함수를
반복 호출한다.
// 그리고 콜백 함수의 반환값이 true인 요소로만 구성된 새로운 배열을 반
환한다.
// 다음의 경우 numbers 배열에서 홀수인 요소만을 필터링한다(1은 true로
평가된다).
const odds = numbers.filter(item => item % 2);
console.log(odds); // [1, 3, 5]
```

map과 달리 filter가 반환한 새로운 배열은 기존의 배열과 길이가 다를 수 있다.

27.9.5 Array.prototype.reduce

자신을 호출한 배열을 순회하면서 콜백 함수를 호출한다.

콜백 함수의 반환값은 다음 순회에서의 콜백 함수의 첫 번째 인수로 전달된다.

이런식으로 순회하면서 최종적으로는 하나의 결과값을 반환한다.

```
// [1, 2, 3, 4]의 모든 요소의 누적을 구한다.
const sum = [1, 2, 3, 4].reduce((accumulator, currentValue,
index, array) => accumulator + currentValue, 0);
```

```
console.log(sum); // 10
```

reduce 메서드는 다양한 방법으로 사용할 수 있다.

평균 구하기

```
const values = [1, 2, 3, 4, 5, 6];

const average = values.reduce((acc, cur, i, { length }) =>
{
  // 마지막 순회가 아니면 누적값을 반환하고 마지막 순회면 누적값으로 평
균을 구해 반환한다.
  return i === length - 1 ? (acc + cur) / length : acc + cu
r;
}, 0);

console.log(average); // 3.5
```

최대값 구하기

```
const values = [1, 2, 3, 4, 5];

const max = values.reduce((acc, cur) => (acc > cur ? acc :
cur), 0);
console.log(max); // 5
```

요소의 중복 횟수 구하기

```
const fruits = ['banana', 'apple', 'orange', 'orange', 'app
le'];

const count = fruits.reduce((acc, cur) => {
  // 첫 번째 순회 시 acc는 초기값인 {}이고 cur은 첫 번째 요소인 'ban
ana'다.
  // 초기값으로 전달받은 빈 객체에 요소값인 cur을 프로퍼티 키로, 요소의
개수를 프로퍼티 값으로
  // 할당한다. 만약 프로퍼티 값이 undefined(처음 등장하는 요소)이면
프로퍼티 값을 1로 초기화한다.
```

```
  acc[cur] = (acc[cur] || 0) + 1;
  return acc;
}, {});

// 콜백 함수는 총 5번 호출되고 다음과 같이 결과값을 반환한다.
/*
{banana: 1} => {banana: 1, apple: 1} => {banana: 1, apple:
1, orange: 1}
=> {banana: 1, apple: 1, orange: 2} => {banana: 1, apple:
2, orange: 2}
*/

console.log(count); // { banana: 1, apple: 2, orange: 2 }
```

중첩 배열 평탄화

```
const values = [1, [2, 3], 4, [5, 6]];

const flatten = values.reduce((acc, cur) => acc.concat(cu
r), []);
// [1] => [1, 2, 3] => [1, 2, 3, 4] => [1, 2, 3, 4, 5, 6]

console.log(flatten); // [1, 2, 3, 4, 5, 6]
```

```
[1, [2, 3, 4, 5]].flat(); // -> [1, 2, 3, 4, 5]

// 인수 2는 중첩 배열을 평탄화하기 위한 깊이 값이다.
[1, [2, 3, [4, 5]]].flat(2); // -> [1, 2, 3, 4, 5]
```

중복 요소 제거

```
const values = [1, 2, 1, 3, 5, 4, 5, 3, 4, 4];

const result = values.reduce(
  (unique, val, i, _values) =>
    // 현재 순회 중인 요소의 인덱스 i가 val의 인덱스와 같다면 val은
처음 순회하는 요소다.
    // 현재 순회 중인 요소의 인덱스 i가 val의 인덱스와 다르다면 val은
```

```
중복된 요소다.
    // 처음 순회하는 요소만 초기값 []가 전달된 unique 배열에 담아 반
환하면 중복된 요소는 제거된다.
    _values.indexOf(val) === i ? [...unique, val] : unique,
  []
);

console.log(result); // [1, 2, 3, 5, 4]
```

27.9.6 Array.prototype.some

자신을 호출한 배열을 순회하면서 콜백 함수를 호출한다.

콜백 함수 반환값이 단 한 번이라도 참이면 true, 거짓이면 false를 반환한다.

```
// 배열의 요소 중에 10보다 큰 요소가 1개 이상 존재하는지 확인
[5, 10, 15].some(item => item > 10); // -> true

// 배열의 요소 중에 0보다 작은 요소가 1개 이상 존재하는지 확인
[5, 10, 15].some(item => item < 0); // -> false

// 배열의 요소 중에 'banana'가 1개 이상 존재하는지 확인
['apple', 'banana', 'mango'].some(item => item === 'banan
a'); // -> true

// some 메서드를 호출한 배열이 빈 배열인 경우 언제나 false를 반환한
다.
[].some(item => item > 3); // -> false
```

27.9.7 Array.prototype.every

자신을 호출한 배열을 순회하면서 콜백 함수를 호출한다.

콜백 함수 반환값이 모두 참이면 true, 거짓이면 false를 반환한다.

```
// 배열의 모든 요소가 3보다 큰지 확인
[5, 10, 15].every(item => item > 3); // -> true

// 배열의 모든 요소가 10보다 큰지 확인
```

```
[5, 10, 15].every(item => item > 10); // -> false

// every 메서드를 호출한 배열이 빈 배열인 경우 언제나 true를 반환한
다.
[].every(item => item > 3); // -> true
```

27.9.8 Array.prototype.find

자신을 호출한 배열을 순회하면서 콜백 함수를 호출하면서 반환값이 true인 첫 번째 요소를 반환한다.

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// id가 2인 첫 번째 요소를 반환한다. find 메서드는 배열이 아니라 요소
를 반환한다.
users.find(user => user.id === 2); // -> {id: 2, name: 'Ki
m'}
```

```
// Array#filter는 배열을 반환한다.
[1, 2, 2, 3].filter(item => item === 2); // -> [2, 2]

// Array#find는 요소를 반환한다.
[1, 2, 2, 3].find(item => item === 2); // -> 2
```

27.9.9 Array.prototype.findIndex

자신을 호출한 배열을 순회하면서 콜백 함수를 호출하면서 반환값이 true인 첫 번째 요소의 인덱스를 반환한다.

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
```

```
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// id가 2인 요소의 인덱스를 구한다.
users.findIndex(user => user.id === 2); // -> 1

// name이 'Park'인 요소의 인덱스를 구한다.
users.findIndex(user => user.name === 'Park'); // -> 3

// 위와 같이 프로퍼티 키와 프로퍼티 값으로 요소의 인덱스를 구하는 경우
// 다음과 같이 콜백 함수를 추상화할 수 있다.
function predicate(key, value) {
  // key와 value를 기억하는 클로저를 반환
  return item => item[key] === value;
}

// id가 2인 요소의 인덱스를 구한다.
users.findIndex(predicate('id', 2)); // -> 1

// name이 'Park'인 요소의 인덱스를 구한다.
users.findIndex(predicate('name', 'Park')); // -> 3
```

27.9.10 Array.prototype.flatMap

map 메서드를 통해 생성된 새로운 배열을 평탄화한다.

즉, map 메서드와 flat 메서드를 순차적으로 실행한다.

```
const arr = ['hello', 'world'];

// map과 flat을 순차적으로 실행
arr.map(x => x.split('')).flat();
// -> ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']

// flatMap은 map을 통해 생성된 새로운 배열을 평탄화한다.
arr.flatMap(x => x.split(''));
// -> ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
```

```
const arr = ['hello', 'world'];

// flatMap은 1단계만 평탄화한다.
arr.flatMap((str, index) => [index, [str, str.length]]);
// -> [[0, ['hello', 5]], [1, ['world', 5]]] => [0, ['hell
o', 5], 1, ['world', 5]]

// 평탄화 깊이를 지정해야 하면 flatMap 메서드를 사용하지 말고 map 메
서드와 flat 메서드를 각각 호출한다.
arr.map((str, index) => [index, [str, str.length]]).flat
(2);
// -> [[0, ['hello', 5]], [1, ['world', 5]]] => [0, 'hell
o', 5, 1, 'world', 5]
```